# 2.5 Data Structures and Their Operations

OCR H446 · Calculators not allowed · about 110 minutes · Write-in edition

Total Marks

Name: _____ Date: ____ / ____ / _____

**Answer ALL questions. Show all your working.**

**1** A print-queue manager used in a school office in Norwich implements a linear queue, a data structure that operates on a First In First Out (FIFO) basis, using an array of size 5 (indices 0 to 4). Two pointers are used: front (the index of the item at the front of the queue) and rear (the index of the item most recently added). Initially front = 0 and rear = -1 (the queue is empty).

**(a)** State what is meant by the term 'First In First Out (FIFO)'. **(1)**

**(b)** The following operations are performed, in order, on the initially empty queue: enqueue(A), enqueue(B), enqueue(C), dequeue(), enqueue(D) **(5)**

Complete a trace table showing the array contents, front and rear after each operation.

**(c)** Explain one disadvantage of a linear queue, compared with a circular queue, in the context of the array above once index 0 has been vacated by the dequeue in part (b). **(2)**

**(d)** Describe how a circular queue would allow index 0 to be reused once rear reaches the last index of the array. **(2)**

**(Total for Question 1 is 10 marks)**

**2** A student-run society at a university in Bristol keeps its members' ID numbers in a singly linked list, in the order they joined. Each node stores a data field and a pointer field holding the address of the next node; a pointer called head stores the address of the first node, and the last node's pointer field holds a null value.

**(a)** State the purpose of the pointer field in a node of a singly linked list. **(1)**

**(b)** The pseudocode below is intended to traverse a linked list from head and output every data value. Complete the two missing lines, (i) and (ii). **(2)**

```
current <- head
WHILE ..........(i).......... DO
  OUTPUT current.data
  current <- ..........(ii)..........
ENDWHILE
```

**(c)** The list currently holds the ID numbers 10 -> 25 -> 40 (head points to the node holding 10; each arrow represents a pointer to the next node; the node holding 40 has a null pointer). A new node called newNode has already been created holding the value 18. Write pseudocode to insert newNode between the nodes holding 10 and 25, given that a pointer variable current already references the node holding 10. **(3)**

**(d)** State the value of head.next.next.data after the insertion in part (c). **(1)**

**(Total for Question 2 is 7 marks)**

---

**3** The society's linked list from Question 2 now holds 10 -> 18 -> 25 -> 40, with head pointing to the node holding 10 and the node holding 40 having a null pointer.

**(a)** A pointer variable previous already references the node holding 18. Write pseudocode to delete the node holding 25 from the list. **(3)**

**(b)** A member of the society claims: 'Deleting a node from a linked list is always more efficient than deleting an item from the middle of an array.' Explain whether this claim is correct, referring to the number of elements that must be updated or moved in each structure. **(3)**

**(c)** State the Big O time complexity of searching for a given value in an unsorted singly linked list of n nodes, and justify your answer. **(2)**

**(Total for Question 3 is 8 marks)**

**4** A calculator app developed by a startup in Manchester evaluates arithmetic expressions that have already been converted into postfix (Reverse Polish) notation, using a stack. Tokens are scanned left to right: when a number is read, it is pushed onto the stack; when an operator is read, the top two values are popped (the second-popped value, a, is the left operand and the first-popped value, b, is the right operand), the operation a operator b is applied, and the result is pushed back onto the stack.

Consider the postfix expression: 6 2 3 + * 4 -

**(a)** Complete a trace table showing the contents of the stack after each token in the expression has been processed. **(6)**

**(b)** State the final value of the expression. **(1)**

**(c)** Explain why a stack, rather than a queue, is the appropriate data structure for evaluating postfix expressions in this way. **(2)**

**(Total for Question 4 is 9 marks)**

**5** A customer-service call centre in Cardiff models its call-holding system as a circular queue, implemented using an array of size 5 (indices 0 to 4), pointers front and rear, and a count of the number of calls currently held. Initially front = 0, rear = 0, count = 0. To enqueue: if count = 5 the queue is full; otherwise array[rear] <- item, rear <- (rear + 1) MOD 5, count <- count + 1. To dequeue: if count = 0 the queue is empty; otherwise item <- array[front], front <- (front + 1) MOD 5, count <- count - 1.

The following calls are held, in order: enqueue(P), enqueue(Q), enqueue(R), dequeue(), dequeue(), enqueue(S), enqueue(T), enqueue(U)

**(a)** Complete a trace table recording front, rear and count after each of the eight operations. **(8)**

**(b)** State the front-to-rear order of the calls remaining in the queue after all eight operations. **(1)**

**(c)** Explain why the array slot at index 0 can safely be reused for enqueue(U), even though it previously held the value P. **(2)**

**(Total for Question 5 is 11 marks)**

**6** A route-planning tool used by a delivery company in Leicester models six delivery hubs, Amir, Beth, Carlos, Dana, Elif and Farah, and the direct roads connecting them, as an undirected graph. The roads are: Amir-Beth, Amir-Carlos, Beth-Dana, Carlos-Dana, Carlos-Elif, Dana-Farah, Elif-Farah. When visiting the neighbours of a hub, they are always considered in alphabetical order.

**(a)** Perform a breadth-first search (BFS) of the graph starting at Amir, using a queue. Show the changing contents of the queue and state the order in which the hubs are visited. **(4)**

**(b)** Perform a depth-first search (DFS) of the graph starting at Amir, using a stack (or recursion), always exploring the first unvisited neighbour in alphabetical order before backtracking. State the order in which the hubs are visited. **(4)**

**(c)** Explain, with reference to the traversals performed in (a) and (b), one situation in which BFS would be preferred over DFS. **(2)**

**(Total for Question 6 is 10 marks)**

**7** A library index system used in a school in Preston stores book reference numbers in a binary search tree (BST), so that they can be searched efficiently. Starting from an empty tree, the following values are inserted, one at a time, always starting comparisons at the root: 50, 30, 70, 20, 40, 60, 80, 35. The standard BST insertion rule is used: at each node, if the new value is less than the node's value, move left; otherwise, move right; the value is inserted at the first empty position reached.

**(a)** Determine the position of 35 in the completed tree, by comparing it against the root and each node subsequently reached, stating at each step whether the comparison sends it left or right. Hence state the parent of 35 and whether it becomes the left or right child of that parent. **(3)**

**(b)** State the value stored at the root of the completed tree. **(1)**

**(c)** Write out the pre-order traversal of the completed tree. **(3)**

**(d)** Write out the in-order traversal of the completed tree, and state what property of a binary search tree this result demonstrates. **(3)**

**(Total for Question 7 is 10 marks)**

**8** A file-directory viewer developed by a software company in Oxford represents a small folder structure as a binary tree, where each node stores a single-letter identifier. The root node is M. M's left child is H and right child is T. H's left child is D and right child is L. T's left child is S and right child is W. Nodes D, L, S and W each have no children.

**(a)** Write out the result of a pre-order traversal of the tree. **(2)**

**(b)** Write out the result of an in-order traversal of the tree. **(2)**

**(c)** Write out the result of a post-order traversal of the tree. **(2)**

**(d)** Write out the result of a breadth-first (level-order) traversal of the tree, and state the abstract data type that is used to implement this traversal, giving a reason for its use. **(3)**

**(Total for Question 8 is 9 marks)**

**9** A social-network analysis tool built by a data-science team in Sheffield models the friendship connections between five users, Aisha, Ben, Chidi, Priya and Tom, as an undirected graph, where an edge represents a mutual friendship. The friendships are: Aisha-Ben, Aisha-Chidi, Ben-Chidi, Ben-Priya, Chidi-Priya, Priya-Tom.

**(a)** Complete the adjacency matrix for this graph, using 1 to indicate a friendship and 0 to indicate no friendship, with rows and columns ordered alphabetically: Aisha, Ben, Chidi, Priya, Tom. **(4)**

**(b)** Write the adjacency list representation of the same graph. **(3)**

**(c)** State one advantage of using an adjacency list, rather than an adjacency matrix, to represent a sparse graph such as this one (a graph with relatively few edges compared with the number of possible edges). **(2)**

**(Total for Question 9 is 9 marks)**

**10** A software engineer at a customer-records company based in Glasgow is designing a system that must store a large, frequently changing collection of customer records and needs to support fast insertion, deletion and look-up of a record by its unique customer ID.

**(a)** Discuss the suitability of a hash table, a binary search tree and a linked list for this purpose, comparing their typical time complexities for insertion, deletion and look-up, and recommend which structure the engineer should choose, justifying your recommendation. **(9)**

**(Total for Question 10 is 9 marks)**

# Mark scheme · 2.5 Data Structures and Their Operations

## Question 1

(a) the item that has been in the structure the longest (added first) is the first one to be removed (oe) **B1**

**(a) Answer: The item that was added earliest is the first one to be removed.**

(b) enqueue(A): array = [A,_,_,_,_], front = 0, rear = 0 **B1**

(b) enqueue(B): array = [A,B,_,_,_], front = 0, rear = 1 **B1**

(b) enqueue(C): array = [A,B,C,_,_], front = 0, rear = 2 **B1**

(b) dequeue(): removes A, front = 1, rear = 2 **B1**

(b) enqueue(D): array = [_,B,C,D,_], front = 1, rear = 3 **B1**

**(b) Answer: After all five operations: front = 1, rear = 3, and the queue (front to rear) holds B, C, D.**

(c) once an item is dequeued from the front, that array slot (e.g. index 0) becomes unusable in a linear queue, since only rear moves forward on enqueue (oe) **B1**

(c) as more items are enqueued and dequeued, rear keeps advancing towards the last index, so the queue can eventually report as full (no more room after rear) even though earlier slots are actually free, wasting space (oe) **B1**

**(c) Answer: Index 0 becomes wasted space once A is dequeued, because a linear queue only ever moves rear forward; eventually rear reaches the final index and the queue is reported full, even though slots such as index 0 are actually empty.**

(d) the rear pointer wraps around back to index 0 using modulo arithmetic, e.g. rear = (rear + 1) MOD size, once it passes the last index (oe) **B1**

(d) this reuses slots that have already been dequeued (are now outside the front-to-rear range), as long as the queue is not already full, so space is not wasted (oe) **B1**

**(d) Answer: In a circular queue, rear is calculated using rear = (rear + 1) MOD size, so once rear passes the last index it wraps back to index 0; provided that slot has already been freed by an earlier dequeue, it can safely be reused instead of being left wasted.**

## Question 2

(a) it stores the address of (a reference to) the next node in the list, or a null value if it is the last node (oe) **B1**

**(a) Answer: It stores the address of the next node in the list, or a null value if there is no next node.**

(b) (i) current <> NULL (or current ≠ NULL / current is not null) (oe) **B1**

(b) (ii) current.next **B1**

**(b) Answer: (i) current <> NULL (ii) current.next**

(c) newNode.next <- current.next (newNode's pointer is set to reference the node holding 25, i.e. what current currently points to) **M1**

(c) current.next <- newNode (the node holding 10 is redirected to point to newNode) **A1**

(c) correctly performs these two assignments in this order, so that the reference to the node holding 25 is not lost before newNode is linked to it (oe) **B1**

**(c) Answer:**

```
newNode.next <- current.next
current.next <- newNode
```

(d) 25 ft from (c) **B1**

**(d) Answer: 25**

## Question 3

(a) target <- previous.next (references the node to be removed, the node holding 25) **M1**

(a) previous.next <- target.next (bypasses the removed node, linking the node holding 18 directly to the node holding 40) **M1**

(a) resulting list correctly reads 10 -> 18 -> 40 (the node holding 25 is no longer reachable and may be discarded/freed) **A1**

**(a) Answer:**

```
target <- previous.next
previous.next <- target.next
```

(b) the claim is correct in terms of the work needed once the position is known (oe) **B1**

(b) deleting from a linked list only requires updating one pointer (previous.next), regardless of how many nodes are in the list, once the node to delete has been located (oe) **B1**

(b) deleting from the middle of an array requires shifting every subsequent element one position to the left to close the gap, so the number of elements moved depends on the size of the array/list (oe) **B1**

**(b) Answer: The claim is correct: once the node before the target has been found, deleting it from a linked list only involves changing one pointer, whereas deleting from the middle of an array requires shifting every element after it one place to the left to close the gap.**

(c) O(n) cao **B1**

(c) in the worst case every node must be visited in turn, following the next pointers from head until the value is found or a null pointer is reached, since there is no direct/random access to a given position (oe) **B1**

**(c) Answer: O(n), because in the worst case every one of the n nodes must be visited in turn, following pointers from head, since a linked list has no direct access to a given position.**

## Question 4

(a) after 6, 2, 3 are read: stack = [6, 2, 3] (three pushes) **B1**

(a) '+' : pops 3 (b) and 2 (a), computes 2 + 3 = 5, pushes 5: stack = [6, 5] **B1**

(a) '*' : pops 5 (b) and 6 (a), computes 6 * 5 = 30, pushes 30: stack = [30] **B1**

(a) after 4 is read: stack = [30, 4] **B1**

(a) '-' : pops 4 (b) and 30 (a), computes 30 - 4 = 26, pushes 26: stack = [26] **B1**

(a) final stack correctly shown as [26] with all intermediate values consistent **B1**

**(a) Answer: Stack after each token: [6] -> [6,2] -> [6,2,3] -> [6,5] -> [30] -> [30,4] -> [26]**

(b) 26 ft from (a) **B1**

**(b) Answer: 26**

(c) an operator always needs to combine with the operands nearest to it (most recently pushed), which matches a stack's LIFO behaviour (oe) **B1**

(c) a queue would return items in the order they were first inserted (FIFO), which would not give the operator the correct, most recent operands needed for evaluation (oe) **B1**

**(c) Answer: An operator always needs to act on the two most recently seen operands, which is exactly what a stack's LIFO behaviour provides; a queue's FIFO behaviour would instead return the oldest values first, giving the wrong operands.**

## Question 5

(a) enqueue(P): array[0] = P, rear = 1, count = 1 **B1**

(a) enqueue(Q): array[1] = Q, rear = 2, count = 2 **B1**

(a) enqueue(R): array[2] = R, rear = 3, count = 3 **B1**

(a) dequeue(): removes P, front = 1, count = 2 **B1**

(a) dequeue(): removes Q, front = 2, count = 1 **B1**

(a) enqueue(S): array[3] = S, rear = 4, count = 2 **B1**

(a) enqueue(T): array[4] = T, rear = (4+1) MOD 5 = 0, count = 3 **B1**

(a) enqueue(U): array[0] = U (overwriting the freed slot that held P), rear = 1, count = 4 **B1**

**(a) Answer: After all eight operations: front = 2, rear = 1, count = 4, with the queue (front to rear) holding R, S, T, U.**

(b) R, S, T, U ft from (a) **B1**

**(b) Answer: R, S, T, U**

(c) P was removed by an earlier dequeue operation, so it is no longer part of the logical queue and is no longer referenced by front or count (oe) **B1**

(c) a circular queue reuses slots via modulo arithmetic on the rear pointer once they fall outside the current front-to-rear range represented by count, avoiding the wasted-space problem of a linear queue (oe) **B1**

**(c) Answer: P was already removed by the earlier dequeue() call, so index 0 is no longer part of the logical queue; because the queue is circular, rear can wrap back to index 0 using modulo arithmetic and safely overwrite it with U.**

## Question 6

(a) visits Amir, then enqueues its unvisited neighbours in alphabetical order, Beth then Carlos: queue = [Beth, Carlos] **M1**

(a) dequeues and visits Beth before Carlos (FIFO order), correctly enqueuing each hub's new unvisited neighbours as they are found (Dana when processing Beth; Elif when processing Carlos; Farah when processing Dana) **M1**

(a) final visit order: Amir, Beth, Carlos, Dana, Elif, Farah cao **A1**

(a) correctly states that the algorithm terminates once the queue becomes empty, with every hub visited exactly once **B1**

**(a) Answer: Amir, Beth, Carlos, Dana, Elif, Farah**

(b) visits Amir, then immediately visits Beth (its first unvisited neighbour), rather than visiting all of Amir's neighbours first **M1**

(b) correctly continues deep from Beth to Dana, then from Dana to Carlos (Beth already visited), then from Carlos to Elif, then from Elif to Farah, backtracking only once no unvisited neighbours remain **M1**

(b) final visit order: Amir, Beth, Dana, Carlos, Elif, Farah cao **A1**

(b) correctly identifies that DFS uses a stack (explicitly, or implicitly via the recursive call stack), in contrast to the queue used by BFS **B1**

**(b) Answer: Amir, Beth, Dana, Carlos, Elif, Farah**

(c) identifies a suitable use case, e.g. finding the shortest route (fewest roads) between two hubs in this unweighted graph (oe) **B1**

(c) justifies this: BFS explores all hubs at the current distance from the start before moving further away, so the first time the target hub is reached is guaranteed to be via a shortest path, whereas DFS explores one branch fully before backtracking and does not guarantee this (oe) **B1**

**(c) Answer: BFS would be preferred for finding the shortest route (fewest roads) between two hubs, because it visits all hubs at each distance from the start before moving further out, so the first time it reaches the target hub is guaranteed to be via a shortest path, which DFS does not guarantee.**

## Question 7

(a) compares 35 with the root, 50: since 35 < 50, moves left (to the subtree rooted at 30) **M1**

(a) compares 35 with 30: since 35 ≥ 30, moves right (to the subtree rooted at 40) **M1**

(a) compares 35 with 40: since 35 < 40, and 40 has no left child, 35 becomes the left child of 40 cao **A1**

**(a) Answer: 35 becomes the left child of 40.**

(b) 50 **B1**

**(b) Answer: 50**

(c) correctly traverses the left subtree of 50 (rooted at 30) in pre-order: 30, 20, 40, 35 **M1**

(c) correctly traverses the right subtree of 50 (rooted at 70) in pre-order: 70, 60, 80 **M1**

(c) 50, 30, 20, 40, 35, 70, 60, 80 cao **A1**

**(c) Answer: 50, 30, 20, 40, 35, 70, 60, 80**

(d) correctly traverses the left subtree of 50 in-order: 20, 30, 35, 40 **M1**

(d) correctly traverses the right subtree of 50 in-order: 60, 70, 80, giving the full sequence 20, 30, 35, 40, 50, 60, 70, 80 cao **M1**

(d) states that an in-order traversal of a BST always produces the stored values in ascending (sorted) order (oe) **B1**

**(d) Answer: 20, 30, 35, 40, 50, 60, 70, 80. This demonstrates that an in-order traversal of a binary search tree always visits the values in ascending (sorted) order.**

## Question 8

(a) correct method applied: node, then left subtree, then right subtree, at every level **M1**

(a) M, H, D, L, T, S, W cao **A1**

**(a) Answer: M, H, D, L, T, S, W**

(b) correct method applied: left subtree, then node, then right subtree, at every level **M1**

(b) D, H, L, M, S, T, W cao **A1**

**(b) Answer: D, H, L, M, S, T, W**

(c) correct method applied: left subtree, then right subtree, then node, at every level **M1**

(c) D, L, H, S, W, T, M cao **A1**

**(c) Answer: D, L, H, S, W, T, M**

(d) M, H, T, D, L, S, W cao **B1**

(d) a queue **B1**

(d) nodes are added (enqueued) to the queue in the order they are discovered, level by level, and removed (dequeued) in the same order, so the FIFO behaviour of a queue ensures each level is fully visited before moving to the next (oe) **B1**

**(d) Answer: M, H, T, D, L, S, W. A queue is used, because its FIFO behaviour ensures nodes are visited level by level, in the same order they were discovered.**

## Question 9

(a) Aisha's row/column correct: connected to Ben and Chidi only **B1**

(a) Ben's row/column correct: connected to Aisha, Chidi and Priya **B1**

(a) Chidi's and Priya's rows/columns correct: Chidi connected to Aisha, Ben, Priya; Priya connected to Ben, Chidi, Tom **B1**

(a) Tom's row/column correct (connected to Priya only), matrix symmetric about the diagonal, and all diagonal entries 0 (no self-friendships) **B1**

**(a) Answer: Aisha Ben Chidi Priya Tom**
**Aisha 0 1 1 0 0**
**Ben 1 0 1 1 0**
**Chidi 1 1 0 1 0**
**Priya 0 1 1 0 1**
**Tom 0 0 0 1 0**

(b) Aisha: [Ben, Chidi] **B1**

(b) Ben: [Aisha, Chidi, Priya]; Chidi: [Aisha, Ben, Priya] **B1**

(b) Priya: [Ben, Chidi, Tom]; Tom: [Priya] **B1**

**(b) Answer: Aisha: [Ben, Chidi]; Ben: [Aisha, Chidi, Priya]; Chidi: [Aisha, Ben, Priya]; Priya: [Ben, Chidi, Tom]; Tom: [Priya]**

(c) an adjacency list uses less memory for a sparse graph, since it only stores the edges that actually exist (oe) **B1**

(c) an adjacency matrix always requires space proportional to the square of the number of vertices ($n^2$), regardless of how few edges are actually present, which wastes space for a sparse graph (oe) **B1**

**(c) Answer: An adjacency list uses less memory, because it only stores the edges that actually exist, whereas an adjacency matrix always needs $n^2$ storage regardless of how sparse the graph is.**

## Question 10

(a) **L1** Limited or generic statements are made about one or more of the structures, with little or no reference to time complexity, or an unjustified/unsupported recommendation is given. **1-3**

(a) **L2** A generally sound discussion compares at least two of the three structures, with mostly correct time complexities and some relevant trade-offs, and gives a recommendation with partial or limited justification. **4-6**

(a) **L3** A thorough, well-structured discussion accurately compares all three structures, using correct Big O notation for insertion, deletion and look-up, evaluates relevant trade-offs (e.g. hash-table collisions/worst case, BST balance, linked-list linear search), and gives a clearly justified recommendation tied to the stated requirements. **7-9**

**(a) Indicative content:**

- Hash table: average-case O(1) for insertion, deletion and look-up, given a good hash function and a reasonably low load factor; worst-case performance can degrade towards O(n) if there are many collisions (e.g. a poor hash function, or a high load factor causing clustering); a hash table does not keep records in any particular order.
- Binary search tree: average-case O(log n) for insertion, deletion and look-up if the tree remains reasonably balanced; worst case degrades to O(n) if the tree becomes unbalanced (for example if customer IDs are inserted in already-sorted order), unless a self-balancing variant is used; a BST does maintain records in sorted order, which is useful if ordered/range-based access is also needed.
- Linked list: O(1) insertion at a known position such as the head, but O(n) look-up and O(n) deletion, since the list must first be traversed from the head to locate the record, as there is no direct/random access to a given position.
- For this scenario, a hash table is generally the most suitable choice, because fast insertion, deletion and look-up by a unique ID are all explicitly required, the customer ID is well suited to being used as (or converted into) a hash key, and there is no stated requirement to keep records in a particular order.
- A valid alternative recommendation, such as a self-balancing binary search tree, can be credited if it is justified with accurate reasoning, for example if ordered traversal of customers by ID were also a requirement of the system.

**(a) Answer: A hash table is the most suitable choice, since it offers average-case O(1) insertion, deletion and look-up by unique ID (versus O(log n) for a balanced BST and O(n) for look-up/deletion in a linked list), and no ordering requirement is stated that would favour a BST.**